Enshroud Project
Smart Contract Audits
Resolution Report

Enshroud Dev Team
19 May, 2025

The SourceHat Team performed a comprehensive audit of all 6 of our smart contracts and prepared a report in October 2024, which can be viewed here.

This document describes the mitigation our team performed on each of the Findings identified by the SourceHat auditors. These corrections were deployed on the Sepolia testnet in updated contracts on 28 January, 2025 (see table below). We shall take the issues in the same order they were presented in the audit report.

Finding #1: [DAOPool contract]

Description: *When a claim for deposit trade is created, an open offer is created for the user offering the shares for a claim, but when the trade is executed, the share amount is decremented from the open offer of the user providing the tokens.*

Resolution Actions:

Because the functionality was likely of limited utility (especially without integrated market price feeds), the entire mechanism of offering claimable tokens to other users in exchange for $ENSHROUD has been removed. This moots the error described by this Finding, and also significantly simplifies the DAOPool contract.

Finding #2: [DAOPool contract]

Description: *When claiming Enshroud tokens, the total staked amount is excluded from the total claimable tokens to prevent user's staked balances from being withdrawn as rewards; however, deposited (unstaked) Enshroud tokens are not excluded.*

Risk/Impact: *Users' deposited but unstaked ENSHROUD balances will be distributed as rewards.*

Recommendation: *The contract's total deposited ENSHROUD should be excluded from claimable rewards.*

Resolution Actions:

The auditors' Recommendation was implemented. This was done by adding a new global variable:

```
uint256 public depositedUnstaked;
```

used to track the number of $ENSHROUD tokens deposited to the contract but not staked. This variable is incremented during deposits and unstakes [in functions *depositRegular()*, *depositWithPermit()*, *unstake()*, *deposit()*, and *depositWithLocking()*], and decremented during withdrawals and stakes [in functions *withdraw()* and *stake()*]. The *depositedUnstaked* total is then excluded from computation of claimable yields for $ENSHROUD tokens (along with *totalStake*) in the functions *claimYield()* and *claimTokenYieldAsENFTs()*.

Finding #3: [DAOPool contract]

Description: *The claimed mapping for ETH is used to ensure that a user has not claimed before making a claim offer as ETH is intended to be claimed on any type of claim, but the claimTokenYieldAsENFTs() function does not claim ETH.*

```
if (BLOCKS_PER_EPOCH > (block.number - claimed[msg.sender][address(0)]))
        revert AlreadyClaimedThisEpoch();
```

Risk/Impact: *An offeror can claim yield in every available token excluding ETH, then offer a claim. The token offeror will still be permitted to accept the trade, but the claim offeror will only lose a portion of their ETH rewards while still receiving the full value from their other claims. This can be exploited by a user with two addresses by executing offers to another address owned by them in order to execute multiple token claims using the same shares.*

Recommendation: *The claimTokenYieldAsENFTs() function should automatically distribute ETH rewards and update a user's claimed mapping for ETH.*

Resolution Actions:

While the Risk scenario identified by the auditors was obviated by the removal of the offer claim functionality, the Recommendation was nevertheless implemented. The function *claimTokenYieldAsENFTs()* now also claims ETH, following the loop on the passed ERC20 token contracts, analogous to the procedure used in *claimYields()*.

Finding #4: [DAOPool contract]

> ***Description:*** *A user can stake and lock another user's tokens on behalf of them at any time.*
>
> ***Risk/Impact:*** *A user's deposit balance can be locked by another user at any time, preventing them from withdrawing for at least one week.*
>
> ***Recommendation:*** *Users should only be permitted to stake their own deposit balance, or staking approval functionality should be added.*

Resolution Actions:

The auditors' Recommendation was implemented. The *stake()* function was modified as follows:

```
if (_staker != msg.sender) revert NotDepositOwner();
```

This prevents the malicious locking scenario described.

Finding #5: [DAOPool contract]

> ***Description:*** *When staking on behalf of another user, the specified user's staked tokens are delegated to the caller's delegate instead of the staker's.*
>
> ***Risk/Impact:*** *Voting power may be incorrect or break due to incorrect delegation of voting power.*
>
> ***Recommendation:*** *Users should only be permitted to stake their own deposit balance.*

Resolution Actions:

The auditors' Recommendation was implemented as a consequence of the resolution of Finding #4, because users are no longer permitted to stake on behalf of others, and the internal function *_updateDelegatedVotingPower()* only manipulates the delegated user of *msg.sender*.

Finding #6: [DAOPool contract (sic, actually MVOStaking)]

Description: *Users cannot withdraw from their staked MVO balance if they have an active Timelock.*

```
if (_remaining != 0 && _amount > _unlockedAndWithdrawable)
        revert AmountLargerThanWithdrawable();
```

Risk/Impact: *If a malicious user withdraws their Timelock balance to another user's MVO address, the recipient will not be permitted to withdraw from their MVO until the Timelock has completed, regardless of if they have an existing unlocked staking balance.*

Recommendation: *Timelocked balances should not prevent users from withdrawing their non-timelocked balances.*

Resolution Actions:

This finding was erroneously assigned to the DAOPool contract, but the referenced code actually occurs in the MVOStaking contract. The fix was to make the *withdraw()* function support treating tokens outside of the Timelock separately from tokens (withdrawable or not) within the Timelock. The altered code now looks like this:

```
// _unlockedAndWithdrawable = unlocked - withdrawn, which cannot be negative;
will return 0 if no timelock

uint256 _unlockedAndWithdrawable = _getUnlocked(msg.sender) -
      (timelocks[msg.sender].totalAmount - _remaining);

// in addition to withdrawable from timelock, user can withdraw any tokens
which were added via stake() or stakeWithPermit()

uint256 _neverLocked = idToMVO[_mvoID].stakingQuantum - _remaining;

// if msg.sender has active timelock, revert if _amount is greater than the
unlocked & not yet withdrawn amount + any separately staked unlocked

if (_amount > _unlockedAndWithdrawable + _neverLocked)

      revert AmountLargerThanWithdrawable();

// subtract _amount first from the struct's remainingAmount

if (_unlockedAndWithdrawable >= _amount)

      timelocks[msg.sender].remainingAmount -= _amount;

else

            timelocks[msg.sender].remainingAmount -=
_unlockedAndWithdrawable;

            // (the rest will come from separately-staked unlocked tokens)
```

These changes mean that even if someone maliciously “gifts” another user with a Timelock, the recipient can nevertheless withdraw their own separately deposited tokens (as well as any withdrawable tokens in the donor’s Timelock, which are deposited as unlocked tokens by the TimelockManager).

Finding #7: [TimelockManager contract]

Description: *The IMVOStaking_Timelock stakeTimelock() function declaration used by the Timelock contract contains parameters in incorrect order.*

```
interface IMVOStaking_Timelock {
        function stakeTimelock(
                string calldata mvoID,
                uint256 amount,
                address staker,
                uint128 releaseStart,
                uint128 releaseEnd
        ) external;
}
```

Timelock stakeTimelock function:

```
function stakeTimelock(
        string calldata _mvoID,
        address _staker,
        uint256 _amount,
        uint128 _releaseStart,
        uint128 _releaseEnd
)
```

Risk/Impact: The Timelock contract's withdrawTimelockToMVOStaking() function will fail if a user has an active Timelock.

Resolution Actions:

This problem was also discovered during contract debugging. The fix was applied in the MVOStaking contract, making the function declaration read (swapping order of 2nd and 3rd parameters):

```
function stakeTimelock(
        string calldata _mvoID,
        uint256 _amount,
        address _staker,
        uint128 _releaseStart,
        uint128 _releaseEnd
)
```

Finding #8: [EnshroudProtocol contract]

Description: *A user can provide any token for deposit, but fee-on-transfer tokens are not supported.*

Risk/Impact: *User balances will be larger than their deposited amounts. As a result, user withdrawals may fail or result in a loss of other users' deposited funds.*

Recommendation: *The team should add support for fee-on-transfer tokens, or add a token whitelist.*

Resolution Actions:

Support was added for tokens that are fee-on-transfer and/or deflationary. This was accomplished by checking the EnshroudProtocol's balance in the token before and after the deposit event. Example code from the function *_depositTokens()*:

```
uint256 _ourBalanceBefore =
IERC20Permit_EnshroudProtocol(_tokenContract).balanceOf(address(this));

_tokenContract.safeTransferFrom(_depositor, address(this), _amount);

uint256 _ourBalanceAfter =
IERC20Permit_EnshroudProtocol(_tokenContract).balanceOf(address(this));

require(_ourBalanceAfter >= _ourBalanceBefore);

_netAmount = _ourBalanceAfter - _ourBalanceBefore;
```

This procedure is also utilized in DAOPool:*claimTokenYieldAsENFTs()* when depositing claimed yields in a particular token to the EnshroudProtocol contract on behalf of the claiming user.

Finding #9: [Crowdsale contract]

Description: *The minWei variable used to prevent users from sending less than the minimum token price is set to the initial token price upon deployment, but is not increased to match the price increase from Tier increments.*

Risk/Impact: *If a user provides less than the minimum price of a token after the Tier has increased, they will not receive any tokens and lose their provided ETH.*

Recommendation: *The minWei variable should be increased on every Tier increment.*

Resolution Actions:

The auditors' Recommendation was implemented. The function *_incrementTier()* now includes:

```
minWei *= 2;
```

along with removing the *immutable* modifier from minWei.

Finding #10: [EnshroudProtocol and MVOStaking contracts]

Description: *The RequestApproved() function requires three Admins to approve certain administrative updates before it can be executed. This is done by approving a certain address; however, there is no restriction on which administrative function the proposed address will be used for.*

```
function _isRequestApproved(
        address _reqAddr,
        bytes8 _caller
) internal returns (bool) {
        bytes8 _default;

        if (updateRequested[_reqAddr].requester1 == _default)
                updateRequested[_reqAddr].requester1 = _caller;
        else if (
                updateRequested[_reqAddr].requester1 != _default &&
                updateRequested[_reqAddr].requester2 == _default &&
                _caller != updateRequested[_reqAddr].requester1
        ) updateRequested[_reqAddr].requester2 = _caller;
        else if (
                updateRequested[_reqAddr].requester1 != _default &&
                updateRequested[_reqAddr].requester2 != _default &&
                _caller != updateRequested[_reqAddr].requester1 &&
                _caller != updateRequested[_reqAddr].requester2
        ) {
                return true;
        }
        return false;
}
```

Risk/Impact: *If two Admins call a function to update an address, such as setMVOStakingAddress(), a third Admin could call updateDaoPool(), providing the address intended to be used as the MVOStaking contract. This function would execute as the address had been given two approvals.*

Recommendation: *The team should ensure that address approvals are function-specific.*

Resolution Actions:

The auditors' Recommendation was implemented. This was done by means of adding new enums to both contracts, adding a parameter of this type to *_isRequestApproved()*, with a matching field in *Requests {}*, and passing the appropriate value in all calling code. In EnshroudProtocol the enum is:

```
enum ApprovalTypes { MVOStaking, GreyList, Auditor, Treasury, DaoPool }
```

and in MVOStaking the enum is:

```
enum ApprovalTypes { MinStake, Protocol }
```

With matching changes in *_isRequestApproved()*, this prevents the scenario described from occurring.

Finding #11: [EnshroudProtocol contract]

> ***Description:*** *User points can be artificially inflated through multiple deposits of valueless tokens or using a malicious contract. As user points do not serve any purpose on-chain, this has no impact within the scope of this audit.*
>
> ***Recommendation:*** *The team should consider adding a token whitelist or updating the user points logic.*

Resolution Actions:

The *userPoints* mapping in the EnshroudProtocol contract was meant to keep track of which users have performed fee-earning protocol interactions (deposits or withdrawals of ETH or ERC20 tokens), as well as the total number of such points across all users (via the *totalUserPoints* variable). As noted by the auditors, this is done without regard to the relative value of any given token, or the quantity of the deposit or withdrawal (note fees are percentage-based). The intent was that scores could be used to calculate pro-rated eligibility for a future airdrop of $ENSHROUD tokens, plus allowing users to track their eligibility.

The Team has decided that a better mechanism would be to search for *DepositETH, DepositERC20, WithdrawETH*, and *WithdrawERC20* events. This would allow the token whitelist to be applied to screen out valueless tokens and/or trivial amounts, while also restricting the domain for events to a time period between certain specific start and end blocks. The set of eligible users, as well as their relative contributions to fee revenues during the subject time period, could then be determined more equitably.

While this has not yet been done in the latest test version, it's possible that the *userPoints* mapping will be removed from the contract prior to live deployment, simply to save gas.

As stated above, new versions of the smart contracts have been deployed, with verified sources.

Table 1: Revised Smart Contracts

Contract Name	Address	Block Deployed
EnshroudToken	0x7bA32A25E01F24b6c4D94783a3C154840268344d	7591976
Crowdsale	0x2c3D8d4597cd6bbFF222235eAb745FF7B997e84f	8329549
EnshroudProtocol	0xc353a083050bFF5c89cD123B827fDd6a0314Ac8a	8329553
DAOPool	0xc056f8280999ee0c5A99305c522978873422256B	8329557
TimelockManager	0x4ba56e866177cf38AAe3F49e0763aFE21695bE2e	8329631
MVOStaking	0xbe88Fe067D7D363C6f9435278ba2fa0d6fBF481A	8329635

The latest source code is available from https://sepolia.etherscan.io for all contract addresses.

Vulnerability Analysis Finding: Centralization of Control {WARNING}

- *The Admins and mentioned Roles have the permissions described above, including the ability to mint various tokens.*
- *Admins have the ability to update various addresses throughout the platform.*

Team Comment:

The ability to update various addresses is intentional, and required in order to support individual updates to contracts without redeploying all of them. In particular the EnshroudProtocol and MVOStaking contracts may be updated in future to V2, V3 etc. to add additional features.

The minting of $ENSHROUD tokens will be controlled by the EnshroudDAO, but the project's tokenomics dictate that 45M will be minted overall, in three groups of 15M each: to Founders/Team (via Timelocks), to Users (via a future airdrop), and to MVO operators (via the MVOStaking contract's minting rewards logic). All other tokens (an indeterminate number) will be minted to purchasers by the Crowdsale contract, and thus in response to market demand. Following the airdrop, it is possible that the EnshroudDAO will order the minting admins burned via EnshroudToken:*updateMinterStatus()*.

It is the Team's opinion that this modest degree of centralization is both necessary and tolerable.